Optimising Quantified Expressions in the Automated Constraint Modelling Tool TAILOR

Andrea Rendl and **Ian Miguel** and **Ian P. Gent**
University of St Andrews, UK
email: {andrea,ianm,ipg}@cs.st-andrews.ac.uk

Abstract

Constraint Programming (CP) is a powerful technique for solving large-scale combinatorial (optimisation) problems. However, CP is often inaccessible to users without expert knowledge in the area, precluding its widespread use. One of the key difficulties lies in formulating an effective constraint model of an input problem for input to a constraint solver. Automated Constraint Modelling tools address this issue by providing assistance in model formulation. TAILOR is one such modelling tool. It incorporates a number of highly effective *model optimisations*, which can compensate for a wide selection of poor modelling choices that novices (but also experts) often make. In the context of TAILOR, this paper presents new constraint model optimisation techniques concerned with optimising quantified expressions, constructs that are commonly used in high-level constraint modelling languages, similar to for-loops in programming languages. Our experimental results show that quantified expression optimisations can reduce solving time very considerably.

Introduction

Constraint Programming (CP) is a powerful technique for solving large-scale combinatorial (optimisation) problems. Unfortunately, CP is often inaccessible to novice users, precluding its widespread use. One of the principal reasons that expertise is required is the so-called *modelling bottleneck*: the task of formulating an *effective* constraint model (one that can be solved efficiently) for input to a constraint solver. In order to address this problem, recent research in automated constraint modelling has begun to provide assistance in formulating constraint models.

TAILOR is one such automated constraint modelling tool. It embodies a simple 'model-and-solve' approach, where the user can formulate a straightforward model and - at the push of a button - send it to be solved by a number of different constraint solvers, without requiring any knowledge of the underlying solver. Central to TAILOR's ability to obtain good performance from relatively naive input models are its suite of *model optimisations*, which can compensate for a wide selection of poor modelling choices that novices (and some experts!) often make.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In previous work (Rendl et al. 2009) we have presented several of these optimisation techniques, which can provide impressive speedups. In this paper, we extend this work with *quantified expression optimisations*. Quantified expressions in constraint models perform a similar function to loop constructs in general programming languages. Just as in programming, there are a number of potential pitfalls in using quantified expressions that can lead an inexperienced user to produce an inefficient model. We will demonstrate that these pitfalls can be avoided automatically through the use of the optimisations described in this paper, which can lead to a substantial increase in model performance.

Automated Constraint Modelling

In CP, problems are formulated as Constraint Satisfaction Problems (CSPs), which consist of a finite set of variables V, a finite set of constraints, where each constraint is an arbitrary relation over a subset of variables $V_k \subseteq V$, and each variable $v_i \in V$ is assigned a finite (integer) domain D_i. As an example, consider the n-queens problem (Nadel 1990): placing n queens on an $n \times n$ chessboard such that no two queens attack another. A naive constraint model is given in Example 1, which contains n variables (contained in the array 'q'), where each variable represents the row-position of a queen and thus ranges over values $(1..n)$ (line 2). The first constraint (line 4) states that no two queens may be in the same column: the global constraint '**alldifferent**(q)' states that every variable in 'q' has to take a different value. The second and third constraints (line 6-7,9-10) disallow two queens positioned on the same NW- and SW-diagonal, respectively.

Example 1. Naive n-Queens problem model formulated in solver-independent constraint modelling language ESSENCE′

```
0  given n: int(1..)  $ number of queens $
1  find  q: matrix indexed by [int(1..n)] of int(1..n)
2  such that
3  $ queens positioned in different columns $
4     alldifferent(q),
5  $ no two queens on same NW-SE diagonal $
6     forall i,j:int(1..n) .
7        (i!=j) => (q[i]+i != q[j]+j),
8  $ no two queens on same SW-NE diagonal $
9     forall i,j:int(1..n) .
10       (i!=j) => (q[i]-i != q[j]-j)
```

Typically, constraint problems are modelled as problem *classes* where parameters scale the problem (like parameter n in n-queens). A problem *instance* (CSP) is obtained by specifying the parameters of a class. Constraint solvers solve individual instances.

The main difficulty in constraint modelling is to determine a model of high quality, i.e. will be solved efficiently. For instance, the n-queens model above is inefficient compared to other n-queens constraint models (Nadel 1990), however, it is an intuitive formulation and hence very likely to come from a novice. The aim of *automated constraint modelling* tools is to assist novices with the formulation of an appropriate constraint model.

Three major tools exist to date. First, the O'CASEY system (Little et al. 2003) uses case-based reasoning to store, retrieve and reuse constraint programming experience. In particular, problems are paired with model instances to form a 'case'. Second, CONACQ (Bessière et al. 2007) is a SAT-based version space algorithm to acquire constraint networks: given a set of variables (with associated domains), solutions and non-solutions from the user, CONACQ generates a constraint model by applying machine learning. Third, CONJURE (Frisch et al. 2005) is an automated refinement system, that, given an abstract problem specification, returns a set of constraint models derived from the specification, using non-deterministic refinement rules.

All three tools aim at *generating* an efficient constraint model using either previous knowledge (O'CASEY), variables combined with solutions and non-solutions (O'CASEY) or an abstract problem specification (CONJURE). The generation of a constraint model is an essential step in automated constraint modelling, though not enough: automatically generated models often (1) benefit from further enhancements and (2) need to be translated to solver input format in order to be solved. These two steps, which are incorporated in TAILOR, are performed *after* generating the constraint model, and hence TAILOR constitutes an important *addition* to existing automated modelling tools.

TAILOR has evolved from a simple translator (Ian P. Gent and Rendl 2007) to a powerful automated modelling tool which has inspired other important tools, like the MiniZinc-FlatZinc translator, `mzn2fzn` (Nethercote et al. 2007), that now incorporates some of our optimisation techniques, such as common subexpression elimination (CSE).

TAILOR in a Nutshell

TAILOR is an interactive modelling assistant that is freely available at TAILOR's website (Rendl 2009). It provides a graphical user interface in which the user can model, automatically enhance and solve her problem (using an external solver). Most notable, TAILOR can process both problem instances and *classes*. Processing whole problem classes can be beneficial for two reasons: first, it delivers enhancement of a whole problem class (which needs not be repeated for every instance). Second, since many solvers are libraries of programming languages where problems are formulated as programs and parameters instantiated at runtime, problems can be formulated as classes. To the best of our knowledge, TAILOR is the only existing tool that can translate constraint classes. As input, TAILOR takes constraint models formulated either in modelling language ESSENCE′ (Frisch et al. 2008) or XCSP 2.1 (Roussel and Lecoutre 2008) and can perform three tasks (see Fig. 1):

1. Generation of Solver Input. TAILOR can generate input for solvers MINION (text) and Gecode (C++,FlatZinc), an important contribution: formulating solver input requires a lot of solver background knowledge and is a tedious task, since often instances contain 10,000s of constraints/variables. It is far more convenient to model problems using a *solver-independent* modelling language, such as ESSENCE′, that provides abstract constructs, such as quantification (in comparison, it is much more convenient to write a program in a high-level language, such as Java, than in machine code). Therefore, a compilation from high-level model to low-level solver input (which TAILOR often performs in fractions of a second), provides important assistance to the user, allowing the expert to adapt the generated instance to his needs (e.g. changing search heuristics).

2. Automated Model Optimisations. TAILOR performs a set of light-weight model optimisations, that can deliver solving time speedups of over a magnitude (Rendl et al. 2009). A novel set of optimisation techniques is that concerned with *optimising quantified expressions*, as presented in the following sections. Most importantly, those optimisation techniques are applicable to both problem instances and classes. TAILOR can return an enhanced model in in intermediate format (ESSENCE′ or FlatZinc) or solver format.

3. Complete Control of the Solving Process. TAILOR's most attractive service is the control of the whole compilation/optimisation/solving procedure: given a constraint instance, TAILOR generates enhanced solver input for a selected solver (MINION or Gecode), executes the solver externally, and returns solutions (if any exist) in either ESSENCE′ or FlatZinc format (since solvers often return solutions in a format difficult to read). Therefore, the user only requires the solver executable, but no knowledge about its features, its format, or how to execute it.

In summary, TAILOR can assist both expert and novice in modelling/solving constraint models by compiling, en-

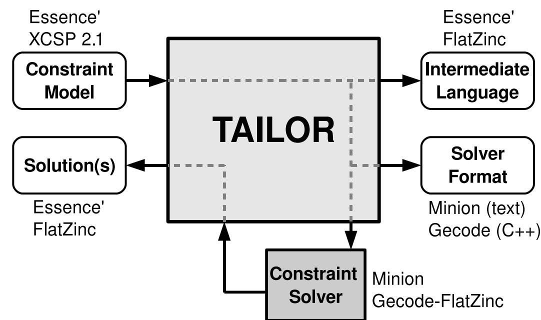

Figure 1: **Tailor's Features**. Given a constraint instance/class, TAILOR can: (1) produce an enhanced model in intermediate language (ESSENCE′ or FlatZinc), (2) generate solver input for MINION (text-format) or Gecode(C++), (3) execute Gecode or MINION and return solutions in ESSENCE′ or FlatZinc

hancing and solving them automatically, which it has already demonstrated in (Loewenstern 2009). In the following, we present new model optimisation techniques that are concerned with optimising quantifications.

Quantified Expressions

Quantified Expressions (denoted *quantifications*) are a very powerful means in constraint modelling languages to express a set of constraints in a compact way. We consider quantifications of the form

$$\varphi\ i_1, ..., i_m : int(lb..ub).\ \ E(i_1, ..., i_m)$$

where $\varphi \in \{\forall, \exists, \sum\}$ is a quantifier that ranges over m quantifying variables $I = \{i_1, \ldots, i_m\}$, each defined over the finite integer range $int(lb..ub)$ (or D for brevity) where $lb \leq ub$, and $E(i_1, \ldots, i_m)$ (or E_I) denotes an expression that is quantified over $\{i_1, \ldots, i_m\}$. Note, that for brevity, we mainly denote quantifications by $\varphi_I D.E_I$.

Similarly to for-loops in program code, quantifications can include redundancies, in particular when formulated by novices. Typically, the negative effect of redundancies increases with the size of the quantifying domain and the number of quantifying variables, i.e. with $m(ub-lb+1)$, which can be vast in large instances. The elimination of redundancies in quantifications is therefore vital.

Addressing Redundancies from Weak Guards

A *guard* B for an expression E is a Boolean expression that has to hold in order to enforce E, e.g. $B \Rightarrow E$. Guards are often used in constraint modelling languages to restrict the number of expressions that a quantification yields. For instance, consider again the n-Queens model in Example 1, where the diagonal-constraints use the guard '`(i!=j)`'.

In universal quantifications, guards are typically of the form $\forall_I B_I \Rightarrow E_I$ where every false B_I 'eliminates' the corresponding E_I since *false*$\Rightarrow E$ is evaluated to *true* (which is the identity of conjunction, i.e. of $\forall$). In existential quantifications, guards are typically of the form $\exists_I B_I \wedge E_I$ where every false B_I 'eliminates' the corresponding E_I, since *false*$\wedge E$ is evaluated to *false* (which is the identity of disjunction, i.e. of $\exists$).

If guards are *too weak*, they do not eliminate the symmetry stemming from commutative operators, and hence yield duplicate expressions in the unrolled quantification. For illustration, unrolling the second constraint in the n-queens model yields the following set of constraints:

```
q[1]+1  != q[2]+2,        q[1]+1  != q[3]+3,
q[2]+2  != q[1]+1,        q[2]+2  != q[3]+3,
q[3]+3  != q[2]+2,        q[3]+3  != q[1]+1,    etc
```

Note, that the list of constraints contains *duplicates*, since the guard in the quantification, '`(i!=j)`', does not break the symmetry of '`!=`', which a stronger guard, '`(i<j)`', would.

In general, two different kinds of duplication can arise from weak guards. First, if B is *constant* (like in n-queens), then duplicate *constraints* arise when unrolling the quantification. Otherwise, duplicate *subexpressions* arise after unrolling the quantification, which are eliminated by CSE. Duplicate constraints are easiest eliminated after unrolling a quantification. If all constraints are *ordered* during compilation, duplicates are positioned *consecutively* in the ordered list of constraints and can be detected by testing consecutive constraints for equivalence (in linear time wrt the number of constraints in the problem instance (Frisch, Miguel, and Walsh 2002)). However, this approach is applicable only to *instances*, when all parameters are known (e.g. the diagonal constraints in n-queens can be unrolled only if n is known). Therefore, we present a more general elimination technique where guards are *automatically strengthened*.

Strengthening Guards by Unification

Unification is a means to find substitutions that render different logical expressions equivalent (Russell and Norvig 2003). Unification is applied through the UNIFY algorithm that, given two logical sentenses E_1 and E_2, returns a unifier u (if one exists):

$$\text{UNIFY}(E_1, E_2) = u \text{ where } \text{SUBST}(u, E_1) = \text{SUBST}(u, E_2)$$

where $\text{SUBST}(u, E)$ denotes the result of applying the substitution u to E. As an example, consider the two expressions $(x+i)$ and $(x+3)$ that have the unifier $u = \{3/i\}$, i.e. if i is substituted with (i.e. assigned) 3, then both expressions are equivalent. We can exploit unification to eliminate duplicate constraints in the following way: given a quantification

$$\varphi I : D.B_I \ominus_\varphi E_I$$

where $\varphi \in \{\forall, \exists\}$, B_I is a Boolean guard, $\ominus_\forall \ = \Rightarrow$ and $\ominus_\exists = \wedge$, and E_I is the guarded expression (considered by its expression tree), we define:

STRENGTHEN_GUARD($\varphi I : D.B_I \ominus_\varphi E_I$)

1. If E_I's root node corresponds to a binary commutative operator, goto 2. otherwise stop.
2. Compute the set of unifiers U for the two children of E_I, e_1 and e_2.
3. Search U for unifiers from which we can deduce equivalence of the quantifying variables. For instance, if two unifiers u_1 and u_2 are of the form $u_1 = \{i_k/i_l\}$ and $u_2 = \{i_l/i_k\}$ where $l, k \in \{1..m\}$ and $l \neq k$, then we can deduce that if $i_k = i_l$ then e_1 and e_2 are equivalent. If successful, goto 4, otherwise stop.
4. Add two conditions C_1 and C_2 to the guard in order to break the equivalence(s) between all quantifying variables i_l and i_k whose equivalence renders e_1 and e_2 equivalent, denoted $\bigwedge_{k,l}(i_k = i_l)$:
   (a) C_1: negation of all equivalent pairs, i.e. $\neg(\bigwedge_{i,k} i_k{=}i_l)$
   (b) C_2: conjunction of lexicographical ordering constraints: for each equivalent pair of quantifying variables, a lexicographical ordering is applied to break the symmetry stemming from the commutative operator. For instance, for the pair $(i_k = i_l)$, we get $(i_k \leq i_l)$.
5. Return $\varphi I : D.(B_I \wedge C_1 \wedge C_2) \ominus_\varphi E_I$

Example: Strengthening the Guard in Golomb Ruler As an example, we consider the constraint of the naive Golomb Ruler model from (Smith, Stergiou, and Walsh 1999) that expresses the distinct distances between all ticks:

```
forall i1,i2,i3,i4 : int(1..ticks).
  ((i1>i2) /\ (i3>i4) /\ (i2!=i4)) =>
     (ruler[i1]-ruler[i2] != ruler[i3]-ruler[i4])
```

The Boolean guard is weak, since it will result in duplicate constraints after unrolling the quantification:

```
 (ruler[1] - ruler[2] != ruler[1] - ruler[3]),
 (ruler[1] - ruler[3] != ruler[1] - ruler[2]),   etc
```

Hence, we apply STRENGTHEN_GUARD to the quantification: first, since '!=' is commutative, we compute the set of unifiers for the two subtrees $(ruler[i_1] - ruler[i_2])$ and $(ruler[i_3] - ruler[i_4])$. There are four unifiers:

$$u_1 = \{i_1/i_3 \wedge i_2/i_4\} \qquad u_2 = \{i_3/i_1 \wedge i_4/i_2\}$$
$$u_3 = \{i_3/i_1 \wedge i_2/i_4\} \qquad u_4 = \{i_1/i_3 \wedge i_4/i_2\}$$

From these unifiers we can deduce that $(ruler[i_1] - ruler[i_2])$ is equivalent to $(ruler[i_3] - ruler[i_4])$ if $(i_1 = i_3) \wedge (i_2 = i_4)$ and two conditions are added to the guard:

1. C_1: $\neg((i_1{=}i_3) \wedge (i_2{=}i_4))$, i.e. $(i_1 \neq i_3) \vee (i_2 \neq i_4)$
2. C_2: Lexicographic constraints $(i_1 \leq i_3)$ and $(i_2 \leq i_4)$.

In summary, STRENGTHEN_GUARD returns the following quantification with a strong guard:

```
forall i1,i2,i3,i4 : int(1..ticks).
 ((i1>i2) /\ (i3>i4) /\ (i2!=i4) /\
 ((i1!=i3) \/ (i2!=i4))  /\   $ condition C1 $
  (i1<=i3) /\ (i2<=i4)) =>   $ condition C2 $
        (ruler[i1]-ruler[i2] != ruler[i3]-ruler[i4])
```

The conditions in the new guard need not be simplified (e.g. $(i \neq j) \wedge (i \leq j)$ to $(i < j)$) since they are quickly evaluated in TAILOR. Note, STRENGTHEN_GUARD can be applied to *any* quantified binary commutative expression and hence used to *generate* strong guards.

Moving Loop-invariant Expressions

In some cases, quantified expressions can be reformulated into equivalent, more efficient representations. Therefore, we study equivalent representations involving *loop-invariant* expressions. We call A in $\varphi_I.A \oplus E_I$ loop-invariant, if A is not quantified by any $i \in I$ and there exists an operator $\oplus'$ such that

$$\varphi_I.A \oplus E_I \equiv A \oplus' \varphi_I.E_I \qquad (1)$$

As an example, consider $\forall_I (x{=}0) \vee (y[i]{=}i)$ that contains the loop-invariant expression $(x{=}0)$ and can be reformulated into $(x{=}0) \vee \forall_I y[i]{=}i$ using the law of distributivity. In this case, the latter representation (*outside*-representation) is typically far more efficient than the former (*inside*-representation). First, we notice that the inside-representation introduces *common subexpressions* as soon as the corresponding quantification is unrolled. More specifically, every quantification $\varphi\ i_1,...,i_m{:}int(lb..ub).\ A \oplus E_I$ is unrolled to

$$(A \oplus E_1) \oplus_\varphi (A \oplus E_2) \oplus_\varphi \cdots \oplus_\varphi (A \oplus E_k)$$

where $\oplus_\varphi$ represents the corresponding operation for φ, i.e. $\oplus_\forall{=}\wedge$, $\oplus_\exists{=}\vee$ and $\oplus_\sum{=}+$. The number of unrolled subexpressions, k, depends on if $A \oplus E_I$ is guarded: if unguarded, $k{=}m{*}(ub{-}lb{+}1)$, otherwise $m{\leq}k{\leq}m{*}(ub{-}lb{+}1)$. Evidently, the unrolled quantification contains k occurrences of the loop-invariant expression A, i.e. common subexpressions that are eliminated by CSE. We therefore see that the redundancies from loop-invariant expressions *in* quantifications can be easily eliminated, therefore the inside-representation is not necessarily always worse than the outside-representation.

Original	$(\forall_I A \Rightarrow E_I)$ (inside)	$A \Rightarrow (\forall_I E_I)$ (outside)
Unrolled	$(A \Rightarrow E_1) \wedge \cdots \wedge (A \Rightarrow E_k)$	$A \Rightarrow (E_1 \wedge \cdots \wedge E_k)$
Flat unnested	$a \Rightarrow e_1$,$a \Rightarrow e_2$,$\ldots$, $a \Rightarrow e_k$	$aux \Leftrightarrow e_1 \wedge \cdots \wedge e_k$ $a \Rightarrow aux$
Flat nested	$aux_1 \Leftrightarrow (a \Rightarrow e_1)$ $\ldots$ $aux_k \Leftrightarrow (a \Rightarrow e_k)$ $aux_q \Leftrightarrow (aux_1 \wedge \cdots \wedge aux_k)$	$aux_1 \Leftrightarrow e_1 \wedge \cdots \wedge e_k$ $aux_q \Leftrightarrow (a \Rightarrow aux1)$

Table 1: Moving loop-invariant A *inside/outside* a quantification.

Comparing inside- and outside-Representation

We want to compare the *inside*- with the *outside*-representation wrt its efficiency in a constraint solver for all operators $\ominus$,φ for which Eq. 1 holds. It is essential to consider each representation at the abstraction level in which it is processed by the solver: its *flat* representation. The flat representation is obtained by *flattening* every constraint to the constraints ('propagators') provided by the target solver, which involves introducing auxiliary variables and additional constraints. For instance, '$A \Rightarrow (x{=}y)$' would be flattened to '$(A \Rightarrow aux) \wedge (aux \Leftrightarrow x{=}y)$', introducing auxiliary variable aux. Flattening is a strictly solver-dependent procedure, therefore we do not conduct a *generic* analysis (covering all possible cases of possible propagators wrt arity, etc), but restrict our analysis to solvers that provide n-ary conjunction, disjunction and summation propagators, which holds for most constraint solvers.

For illustration, Tab.1 depicts how $\forall_I(A{\Rightarrow}E_I)$ and $A{\Rightarrow}(\forall_I E_I)$ are flattened for such solvers: first, the quantification is unrolled, and then each unrolled expression is flattened, for which we distinguish two cases: (1) if to-be-flattened expression E is *not* nested in another constraint, E is flattened to a propagator (constraint), otherwise,(2) E is flattened to an auxiliary variable (denoted with lower-case letters in Tab.1). We then compare the number of constraints and auxiliary variables of the resulting flat representations. For instance, the unnested flat representation of $\forall_I(A{\Rightarrow}E_I)$ in Tab.1 consists of k constraints, introducing no auxiliary variables, while flat $A{\Rightarrow}(\forall_I E_I)$ consists of 2 constraints, introducing 1 auxiliary variable. We perform this analysis of flat representations for *every* case for which Eq. 1 holds

and summarise our results in the table below. Note, that $\forall_I(A \wedge E_I) \equiv A \wedge \forall_I E_I$ as well as $\exists_I(A \vee E_I) \equiv A \vee \exists_I E_I$ and $\sum_I(A + E_I) \equiv mA + \sum_I E_I$ are excluded from this comparison, since they have *equivalent* flat representations.

Cases	unnested	nested
$\forall_I(A \vee E_I)$	k aux, k cts	k aux, k+1 cts
$A \vee (\forall_I E_I)$	**2 aux, 2 cts**	**2 aux, 2 cts**
$\forall_I(A \Rightarrow E_I)$	**0 aux, k cts**	k aux, k+1 cts
$A \Rightarrow (\forall_I E_I)$	1 aux, 2 cts	**2 aux, 2 cts**
$\exists_I(A \wedge E_I)$	k aux, $k+1$ cts	k aux, $k+1$ cts
$A \wedge (\exists_I E_I)$	**1 aux, 2 cts**	**2 aux, 2 cts**
$\sum_I(A * E_I)$	k aux, $k+1$ cts	$k+1$ aux, $k+1$ cts
$mA * (\sum_I E_I)$	**1 aux, 2 cts**	**2 aux, 2 cts**

Each column gives the number of constraints (cts) and auxiliary variables (aux) for the respective flat representation where the more efficient representation (wrt solving time) is given in bold face. The overall result is rather suprising: neither representation dominates the other. This is particularly interesting, since the outside-representation would be expected to *generally* outperform the inside-representation. Note, that for brevity, we only analyse the special case where the inside-representation performs better (see Experiments).

Note that it is difficult to make a *generic* statement on which representation is preferable: the representations are not comparable wrt propagation since solvers provide many different propagators. Moreover, we expect that the preferable representation depends on *other* expressions in the problem model: e.g. if the inside-representation shares common subexpressions in the model it might be preferable, if the outside-representation does not.

The observations from this study are integrated into TAILOR that reformulates expressions into the estimated preferable representation (depicted in table above). Furthermore, the user can change these standard settings and select a representation for each quantification type and hence experiment with different representations.

Experimental Results

All our experiments were performed on a Mac Pro 4.2 with 8 GB RAM that contains 2 Quad-Core Intel Xeon 5500 series processors, each 2.26 GHz (hyper-threading off), using TAILORv3.2.0 , Minion 0.9 and Gecode 3.2.2 with Gecode/FlatZinc 3.2.1 and a timeout of 20 minutes. We use default search heuristics in both solvers, first searching on the main decision variables (in their order of definition in the model), followed by auxiliary variables.

Eliminating Duplicate Constraints

We study the effects of duplicate constraints on the naive n-Queens and Golomb Ruler Problem model as discussed in the respective section, comparing models with weak guards to models with strong guards since TAILOR does not yet perform general duplicate elimination, an item of future work. First, we consider the *growth* of constraints (duplicates) with increasing n in both problems for Minion and Gecode, illustrated in Fig. 2(top). The y-axis gives the constraint increase factor when duplicates are not eliminated. The number of duplicates remains fairly the same in n-Queens, while it linearly increases with n in Golomb Ruler (for both solvers).

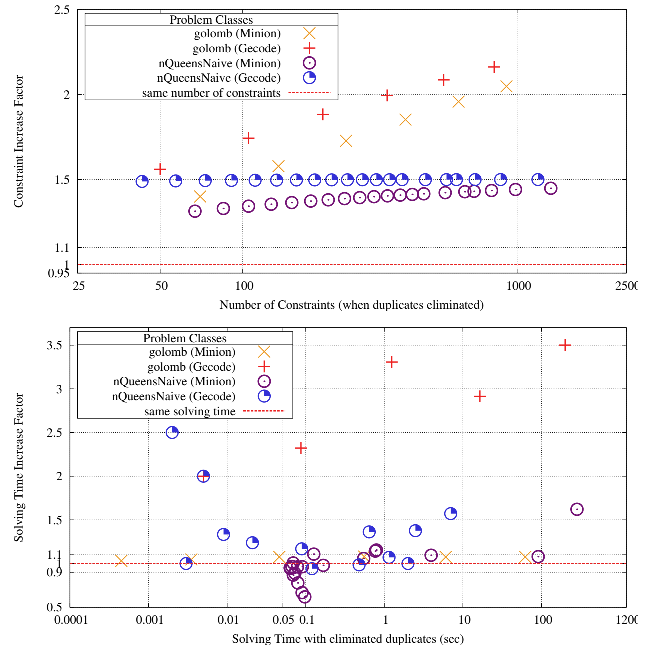

Figure 2: **Eliminating Duplicate Constraints** in naive Golomb Ruler and n-Queens models. **Constraints Increase** with n (top) in instances for Gecode and Minion. **Solving Time Increase** (bottom) in solvers Gecode and Minion when duplicate constraints are not eliminated. For both graphs: y-axis: constraint/solving time increase factor if duplicate constraints are not eliminated.

Second, we consider the effects of duplicates on the solving time in Fig. 2(bottom). The y-axis represents the solving time increase factor when instances contain duplicates. For example, all Golomb instances solved in Gecode lie above y=2, i.e. instances with duplicates are solved in more than twice the time used to solve those without duplicates.

We make three main observations. First, most instances with duplicates are solved using *more* time than those without duplicates (most lie above y=1). Exceptions are some n-Queens instances solved in Minion within 0.005 and 0.1 seconds, hence the differences might stem from external factors, since a 30% solving time difference is very small in this time frame. Second, we observe that Golomb Ruler instances with duplicates perform worse than n-Queens instances with duplicates, probably since the number of duplicates linearly increases with n in Golomb Ruler. Third, duplicate constraints seem to have a far more negative effect in Gecode than in Minion, probably stemming from different approaches in constraint propagations in the solvers.

Moving Loop-invariant Expressions

In this study, we compare inside- and *outside*-representation of $\forall_I A \Rightarrow E_I \equiv A \Rightarrow \forall_I E_I$ in two different models of the Armies of Queens Problem (Smith, Petrie, and Gent 2004) on solvers Gecode and MINION. The differences in instances size is depicted in Fig. 3(top,middle) that illustrating a *reduction* in auxiliary variables (top), and an *increase* of constraints (middle) when using the inside-representation

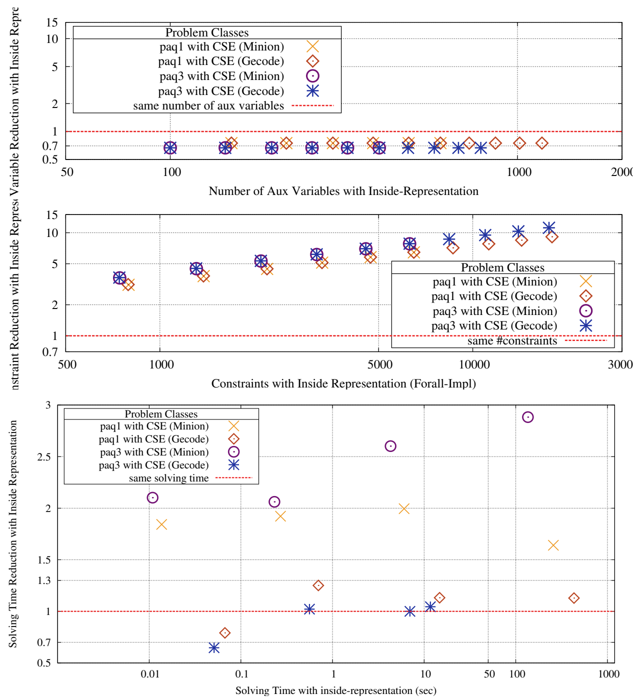

Figure 3: **Moving loop-invariant Expressions**: comparing **Instance Size** and **Solving Time** of different Peaceful Army of Queens instances using inside- and outside-representation. **Auxiliary Variables**(top) and **Constraints**(middle) and **Solving Time**(bottom) for MINION and Gecode. y-axis: increase factor wrt *inside*-representation; e.g. points *above* y=1 depict cases where, using the inside-representation, in (top,middle) aux variables/constraints were *increased*, in (bottom) solving time was reduced.

(exactly the same for both constraint solvers). The bottom graph summarises the comparison in solving time: The x-axis shows the solving time (in sec) for the inside-representation; the y-axis gives the speedup factor using the inside-representation: points above $y = 1$ denote instances that were solved more quickly using the inside-representation, points below depict the opposite.

First, most instances, with the exception of two small instances in Gecode, have been solved quicker using the inside representation. Second, the inside-representation provides a stronger benefit in solver MINION than in Gecode: in MINION the speedup factors goes up to 3, while in Gecode solving time is reduced by moderate 30%. Again, this difference probably stems from the different fashion of posting propagators in each solver (in particular since the inside-representation contains many constraints). Note, that the reformulation of loop-invariant expressions takes little time and has hence no effect on the overall compilation time.

Conclusions

This paper has presented automated model optimisations concerned with improving *quantified expressions* as part of the automated constraint modelling assistant TAILOR, providing two important contributions. First, it presents the automated constraint modelling tool TAILOR, that incorporates a simple 'model-and-solve' approach, from which both CP novice and expert can benefit. TAILOR evolved from a simple translator (Ian P. Gent and Rendl 2007) and now compiles both problem instances and classes to solver input, performing light-weight model optimisations (some presented in earlier work) that can achieve impressive solving time speedups. These model optimisations are extended with our second contribution: *quantification optimisations*, applicable to both instances and classes. As demonstrated on a set of examples, quantification optimisations can compensate for poor modelling choices, resulting in a reduction of solving time to a third. For future work, we plan to extend our investigations of quantification optimisations and other optimisation techniques to provide further enhancement. In summary, TAILOR, and the quantification optimisation techniques it incorporates, constitute a significant contribution to automated constraint modelling.

References

Bessière, C.; Coletta, R.; O'Sullivan, B.; and Paulin, M. 2007. Query-driven constraint acquisition. In *IJCAI 2007*, 50–55.

Frisch, A. M.; Jefferson, C.; Hernández, B. M.; and Miguel, I. 2005. The rules of constraint modelling. In *IJCAI-05*, 109–116.

Frisch, A.; Harvey, W.; Jefferson, C.; Martínez-Hernández, B.; and Miguel, I. 2008. Essence : A constraint language for specifying combinatorial problems. *Constraints* 13(3):268–306.

Frisch, A. M.; Miguel, I.; and Walsh, T. 2002. Cgrass: A system for transforming constraint satisfaction problems. In *International Workshop on Constraint Solving and Constraint Logic Programming*, 15–30.

Ian P. Gent, I. M., and Rendl, A. 2007. Tailoring solver-independent constraint models: A case study with essence' and minion. In *SARA 07: Syposium on Abstraction, Reformulation and Approximation*, 184–199.

Little, J.; Gebruers, C.; Bridge, D. G.; and Freuder, E. C. 2003. Using case-based reasoning to write constraint programs. In *CP-03*, 983.

Loewenstern, A. 2009. Scheduling the cb1000 nanoproteomic analysis system with python, tailor, and minion. In *CP*, 65–72.

Nadel, B. 1990. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert* 5, issue 3:16–23.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. Minizinc: Towards a standard cp modelling language. In *CP 2007*, 529–543.

Rendl, A.; Gent, I. P.; Miguel, I.; and Gregory, P. 2009. Automatically enhancing constraint instances during tailoring. In *SARA 09: Syposium on Abstraction, Reformulation and Approximation*, 120–127.

Rendl, A. 2009. TAILOR: Tailoring constraint models to solvers. http://preview.tinyurl.com/ylgkrqv.

Roussel, O., and Lecoutre, C. 2008. Xml representation of constraint networks format xcsp 2.1. www.cril.univ-artois.fr/CPAI08/XCSP2_1Competition.pdf.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall. chapter Inference in First-Order Logic, 272–319.

Smith, B. M.; Petrie, K. E.; and Gent, I. P. 2004. Models and symmetry breaking for 'peaceable armies of queens'. In *CPAIOR-04*, 271–286.

Smith, B. M.; Stergiou, K.; and Walsh, T. 1999. Modelling the golomb ruler problem. In *Workshop on Non-binary Constraints*.